Pyaella Documentation

Release 0.9

Mat Mathews

July 27, 2015

Contents

Pyaella is a development toolkit designed to make it easy to implement enterprise-class systems. The goal is to help rapidly prototype complex systems, quickly tear-down, build-up iterations of an idea. Pyaella includes tools for metaclasses, metacoding, dependency injection, to simplify desiging business models, logical models, persistence and access.

With this toolkit its possible to rapidly implement a complete backend; defining logical, data and business models with relational integrity and role-based access control.

> **Warning:** This is preliminary documentation and will be constantly changing and improving. Good documentation is hard, and though this project has been under development and active and in production for a few years, the docs are still far behind.

Design Philosophy

The philosophy is simple: Make it painless to start over, to continually improve upon design, usability, stability, and scalability.

Beginnings

This set of tools was initially put together to build custom backends for geospatial analysis and mobile applications.... to leverage the power of PostgreSQL and PostGIS in an agile development environment. To do that we needed to describe and define business domains quickly, using dependency injection and a lightweight DSL. Pyaella's logical domain layer uses dynamically created metaclasses and classes, and SQLAlchemy and GeoAlchemy2 as an ORM to the relational data stored in PostgreSQL.

By defining a single *domain* file describing models, relationships, rules, aspects, and mixins, a new database, an ORM with logical models, business objects and rules, an API, a complete uWSGI supporting Pyramid application can be up in a matter of... well, very quickly.

In Practice

MIGA has been using this code base to prototype and implement numerous backends for projects ranging from gaming analytics to real estate asset management systems.

What Pyaella is not

- Pyaella is not a web framework. It relies on Pyramid
- Pyaella is not a templating system. It relies on Mako
- Pyaella is not a HTML/CSS framework. It relies on either Foundation or MaterializeCSS
- Pyaella is not a NoSQL solution. It relies on PostgreSQL and Redis
- Pyaella is not an ORM or SQL expression toolkit. It relies on SQLAlchemy
- Pyaella is not a messaging system. It relies on RabbitMQ and Celery
- Pyaella is not a remoting / RMI layer. It relies on Pyro

Pyaella, not just for web apps

- Pyaella and Kivy
- Pyaella and Qt
- Support and admin utils for the cloud at Heroku
- Amazon S3 integration AWS
- Geo-spatial logical models, functions, and resources using GEOS
- Complete PostGIS integration
- Easy-to-develop i18n and l10n

- Numpy, SciPy
- Both Heavyweight and lightweight Messaging
- Automated workflow management
- Mobile agents and negociated automation

CHAPTER 1

Quicklook

The core components to Pyaella include a "domain" file that describes the system. This includes information on which metaclasses and superclass to use, how classes are organized and compiled, and any dependency data to be injected at runtime.

Using compilation, install, and refresh tools, the domain file is parsed to create types, classes, models, object-relational mappings, databases, and configures the application runtime, et cætera.

Defining logical models

Here we create a User, with different User Types.

```
User:
    Fields:
        id: Column(BigInteger,
            Sequence('users_id_seq', start=1011011011),
            CheckConstraint('id<11111111111'),
            server_default=text("nextval('users_id_seq')"),
            primary_key=True)
        user_name: Column(String(50), nullable=False, unique=True)
        email_address: Column(String(255), nullable=False, unique=True)
        password: Column(String(128),
            CheckConstraint('"char_length"(password)>126'), nullable=False)
        is_active: Column(Boolean, default=True)
        first_name: Column(String(64), nullable=False)
        last_name: Column(String(64), nullable=False)
        country_code: Column(String(3), nullable=False)
        open_id: Column(String(128))
        auth_code: Column(String(8))
    Relations:
        user_types: relationship('UserXUserTypeLookup')

UserTypeLookup:
    Fields:
        name: Column(String(24), unique=True, nullable=False)
        description: Column(String(64), nullable=False)
    Values:
        name: [
            Sys,
            Dev,
            Admin,
            General
        ]
        description: [
            System Administrator,
```

```
            Developer,
            Administrator,
            General User
        ]
    Rules:
        Unique: [name]

UserXUserTypeLookup:
    Relations:
        user_id: Column(BigInteger, ForeignKey('users.id'), primary_key=True)
        user_type_id: Column(Integer, ForeignKey('user_type_lu.id'), primary_key=True)
```

The above defines a User logical model with numerous *Fields*. In this example we create a SQLAlchemy Column type directly. Any valid *Column* construction is supported, allowing us to define a logical model with parameters, but maintaining a decoupled relationship to an ORM and its underlying persistence layer.

UserTypeLookup is a defined as a "lookup table" style model, where each unique *name* has a description. Each name/description pair defined in this style model is upserted into the table when compiled or upon start-up.

We can associate a *User* with a *UserType* by explicity using an association table. Depending on the case and business rules, this explicit defination may not be required and can be created automatically.

These models are saved to a "domain" file. This domain file is used when designing and implementing the system, generating classes, iterating over implementation details and ideas, and finally when configuring the runtime.

Compiling the logical domain

Pyaella compiles the domain into supporting modules using *pyaellac*.

```
>>> python -m pyaella.metacode.pyaellac -d domain.plr -o models.py
```

The *pyaellac* compilation will output a models.py module that will include decoupled classes that will use the associated domain file during runtime.

```
__autogen_date__ = "2013-08-19 05:43:07.536073"
__schema_file__ = os.path.join(os.path.dirname(__file__), "domain.pyl")
MODEL_SCHEMA_CONFIG = __borg_lex__('ModelConfig')(parsable=__schema_file__)

__all__ = [
    "UserTypeLookup",
    "UserXUserTypeLookup",
    "User"
]

class UserTypeLookup(PyaellaDataModel):
    __metaclass__ = PyaellaDataModelMetaclass

    def __init__(self, base=Base, **kw):
        PyaellaDataModel.__init__(self, base=base, **kw)


class UserXUserTypeLookup(PyaellaDataModel):
    __metaclass__ = PyaellaDataModelMetaclass

    def __init__(self, base=Base, **kw):
        PyaellaDataModel.__init__(self, base=base, **kw)


class User(PyaellaDataModel):
    __metaclass__ = PyaellaDataModelMetaclass
```

```
    def __init__(self, base=Base, **kw):
        PyaellaDataModel.__init__(self, base=base, **kw)
```

You can create the classes and the logical models by hand, but by using *pyaellac* its easier to manage automatic migrations when logical models and data model schemas change and need to be represented in a database.

Once the domain has been compiled, and we have a Pyaella scaffold built, we can edit an *appcfg* file to defile the runtime.

```
App:
    AppId: 1600
    DomainId: 1357913579135791
    UploadDirectory: uploads/
    AssetDepot: s3://asset_depot
    AsyncFamily: Threading
    BackgroundModules: [bgprocs]
Web:
    ScanPackage: miga
    TemplateDir: miga/templates
    StaticDirs:
        assets: miga/assets
        player: miga/player
        css: miga/css
        scripts: miga/scripts
    SiteName: localhost
Resources:
    Database:
        User: postgres
        Password:
        Host: localhost
        Port: 5432
        Schema: pyaella-demo
        CreateTables: True
    ORM:
        PoolSize: 20
        MaxOverflow: 10
        ConvertUnicode: True
        Echo: True
    Contexts: miga.contexts
    Models: miga.models
    Schema: miga/domain.pyl
```

This *appcfg* example above sets up the runtime, and is displayed here for the quicklook example.

Creating the database for the first time

Running the *dbinstall* will create a brand new database called *pyaella-demo* that supports Pyaella, its custom types and functions.

```
python -m pyaella.server.dbinstall -U postgres -O postgres --host localhost --port 5432 --contrib-di
```

The *pyaella.server.dbinstall* takes arguments to specify the user, the host machine, port, and the new database's name. It also takes an argument as the path to the PostgreSQL contribs directory if it exists in a custom location.

The *dbinstall* script uses the appcfg.yaml file.

1.1 Creating a Pyaella application for Pyramid

Installation

Definition

Configuration

Deployment

A Pyramid view_callable example

This is an example snippet from a Miga project

Query for a User of UserType using Pyaella's SQLAlchemySessionFactory and the SQLAlchemy relationship on User for UserXUserTypeLookup

```
with SQLAlchemySessionFactory() as session:
    res_prxy = session.query(~User).filter((~User).user_types.name=='admin').all()
```

The following does an explicit join and lookup to find an admin

```
@view_config(
    name='foundme',
    request_method='GET',
    context='miga:contexts.Geo',
    renderer="foundme.mako")
def foundme(request):
    """ render map of user and nearest droplets
        foundme/lng=10.3031649469030580/lat=43.5311867249821205
    """
    auth_usrid = authenticated_userid(request)

    with SQLAlchemySessionFactory() as session:

        U, UT, UxUTL = ~User, ~UserTypeLookup, ~UserXUserTypeLookup
        user = (session.query(U, UT, UxUTL)
                    .filter(U.email_address==auth_usrid)
                    .filter(UxUTL.user_id==U.id)
                    .filter(UxUTL.user_type_id==UT.id)
                    .filter(UT.name=='admin')).first()
```

Using the default API

Each logical model defined in the domain can be accessible through Pyaella's RESTlike API, an application programming interface that presents method signature-like URIs to GET, POST, PUT, DELETE *entities*.

> **Warning:** Pyaella uses JSON and JSONP exclusively, and does not support "web services" or any XML bloatware

A simple *GET*:

```
http://camp.awesome.sauce.com/m/edit/User/id=5
```

In the above example the *m/* traversal resource is called with the resource name *edit* for the subpath *User/id=5*. User in the logical model, which will be used to retrieve a User entity, using the ORM to access the database, filtering for the User with the id "5". *GETS* are used for retrieval, and not searches.

A search for entities:

```
http://camp.awesome.sauce.com/m/search/User/first_name=Xef
```

The *search* resource will return, if any, all Users with the first name "Xef".

So what did we just do?

Without too much code and writing two configuration files, one a "domain" file and another a "appcfg" file Pyaella built a Pyramid application, a PostgreSQL database, and presents an RESTlike API to the logical and data models. Of course many details were glossed, but most other code and organization required is boilerplate and documented elsewhere.

Roadmap ahead

Pyaella's goal is to mature into a complete stack of development tools to help rapidly cook-up new distributed applications, large infrastructure back-ends, mobile applications, and automation systems; to help the creative process designing, implementing, and testing immediately. With good tools a team can prove a new idea without having to get bogged down with repetitive work, tasks that should be automated, and the headaches of starting from scratch.

Contents:

1.1.1 Installing Pyaella

System / Library Requirements

- Mac OS X 10.9 < or Ubuntu 14.04 LTS
- PostgreSQL 9.2 <
- PostGIS 2 <
- Python 2.7
- file
- libjpeg9 (optional)
- opencv2 (optional)
- mediainfo (optional)

Github

```
git clone http://github.com/migacollabs/Pyaella@master
```

Github into a virtualenv

```
pip install git+https://github.com/migacollabs/Pyaella@master
```

Pyaella is opinionated, and sometimes unapologetically so. It only supports PostgreSQL and other open-source, enterprise-quality solutions like PostGIS. This results in a large dependency tree. When installed using *pip* the following packages are installed automatically.

Python Package Requirements

- Babel==1.3
- Chameleon==2.11
- GeoAlchemy2==0.2.1
- Mako==0.9.1
- MarkupSafe==0.23
- PasteDeploy==1.5.2
- Pillow==2.9.0
- PyYAML==3.10
- Pyaella==0.7.9
- Pygments==1.6
- SQLAlchemy==1.0.6
- Shapely==1.5.9
- WebOb==1.3.1
- boto==2.9.8
- distribute==0.6.31
- docutils==0.12
- futures==2.1.4
- isodate==0.4.9
- jsonpickle==0.4.0
- numpy==1.8.0
- psycopg2==2.6.1
- pyasn1==0.1.7
- pycrypto==2.6
- pygeocoder==1.2.5
- pyramid==1.5b1
- pyramid-debugtoolbar==2.4
- pyramid-mako==0.3.1
- python-geohash==0.8.5
- python-geoip==1.2
- python-geoip-geolite2==2015.0303
- python-magic==0.4.6
- qrcode==3.1.1
- redis==2.8.0
- repoze.lru==0.6
- requests==2.2.1

- rsa==3.1.1
- six==1.9.0
- translationstring==1.1
- venusian==1.0
- waitress==0.8.5
- wsgiref==0.1.2
- zope.deprecation==4.1.2
- zope.interface==4.1.2

Installing on a Mac

Pyaella can use Python and PostgreSQL framework installs, but out-of-the-box it is much easier to use MacPorts. Using *port* will create a separate and clean */opt/* directory and environment with more traditional Unix structure and linking.

Download and install MacPorts. Once it is installed, by following all the instructions, make sure to run the *port selfupdate* command before continuing.

> **Warning:** Because MacPorts still builds the software from source, the Apple Developer Tools are required. These will have to be installed first!
> AND... Its highly dangerous to use multiple Package Managers at the same time, so if you are using Homebrew or something else, stick to that!

Install Python 2.7.x.

```
sudo port install python27
```

Now make sure the Mac Ports Python is selected and active. We don't wnat to be installing packages into the wrong Python environment.

```
sudo port select --set python python27
```

We are going to install Python packages using *pip*. Lets get pip using Mac Ports

```
sudo port install py27-pip
```

Installing bacis unix utilities

```
sudo port install filemagic
```

Installing mediainfo (optional)

Download and install the appropriate version from http://mediaarea.net/en/MediaInfo

Installing JPEG support (optional)

Strangely, adding JPEG encoding and decoding support on a Mac can be troublesome. Its best to get this done immediately before any other dependent libraries need be built.

Download libjpeg9. Unpack the gz and follow the normal compilation instruction. Such as:

```
./configure
make
./sudo make install
```

Installing OpenCV support (optional)

Pyaella uses OpenCV for image and video processing. There are many was to build OpenCV, but the easiest on a Mac is again using MacPorts, with the Python binding variant. This will download, configure, compile and install opencv along with the Python 2.7 bindings.

```
sudo port install opencv +python27
```

> **Warning:** OpenCV and the Python bindings will not be available to a project using *virtualenv* without coping the .so to the virtualenv's lib.

Initializing a PostgreSQL Database

Install the PostgreSQL client libraries.

```
sudo port install postgresql93
```

Install the PostgreSQL server, so a local database is available for development. This is optional but highly recommended.

```
sudo port install postgresql93-server
```

Install the PostGIS extension

```
sudo port install postgis2
```

> **Warning:** Be sure to install PostGIS2 and not PostGIS1.5!

The following commands are the typical PostgreSQL installation process on a Mac OS X machine after using MacPorts:

```
sudo mkdir -p /opt/local/var/db/postgresql93/defaultdb
sudo chown postgres:postgres /opt/local/var/db/postgresql93/defaultdb
sudo su postgres -c '/opt/local/lib/postgresql93/bin/initdb -D /opt/local/var/db/postgresql93/default
```

Note: Starting and stopping the local development database

cd /opt/local/etc/LaunchDaemons/org.macports.postgresql93-server sudo ./postgresql93-server.wrapper start

or -

sudo launchctl load -w /Library/LaunchDaemons/org.macports.postgresql93-server.plist

---

Creating a database

There are a few extras for the database. These include the PostGIS extension, and all the default tables, functions, and types for Pyaella.

You can prepare a Postgres database to be Pyaella 'ready' in a few steps:

The *pyaella.server.dbinstall* module can create a new database, and adds PostGIS and Pyaella support by default. There are two ways to do this, depending on systems administration requirements and preference.

No previous database?

If there is no database set up yet, you can build a Pyaella db using a command like the following:

```
python -m pyaella.server.dbinstall -U postgres -O postgres --host localhost --port 5432 --contrib-di
```

This would create a new database on your locally running Postgres instance, that you have administration access to, as the user 'postgres' and the owner 'posgres'. The *–contrib-dir* argument specifies where the postgis-2 SQL scripts are on your machine, so it can run them, loading a full PostGIS environment.

If PostGIS is not needed or desired, add the *–no-postgis* argument:

```
python -m pyaella.server.dbinstall -U postgres -O postgres --no-postgis --db [DBNAME]
```

If default Pyaella functions, triggers, types, and tables are also not needed or required then also add the *–no-pyaella* argument:

```
python -m pyaella.server.dbinstall -U postgres -O postgres --no-postgis --no-pyaella --db [DBNAME]
```

But what if a database already exists?

You can use the conninfo connection string in the format

postgres://yourname:yourpassword@host/databasename

Export the conninfo as an environment variable *DATABASE_URL* like:

```
export DATABASE_URL=postgres://yourname:yourpassword@host/databasename
```

When you run *dbinstall* don't specify the user *-U* or the owner *-O* or the database *-db* arguments. Pyaella will use your conninfo environment variable to connect and configure the database.

> **Warning:** Using the conninfo environment variable does not allow you to drop the database if it already exists.

1.1.2 Using Pyaella

Here we'll create a demo application called Fusilli. Fusilli will do nothing, but show us how to create a new application from scratch and define a Pyaella logical domain, compile it, package it, and start it up.

Setup an environment

Install virtualenv

Install virtualenv using the *easy_install* for the Python 2.7 in /opt/:

```
sudo easy_install virtualenv
```

Create a new application virtualenv

After installing virtualenv package, create a new virtual environment for the new application:

```
virtualenv --no-site-packages --distribute FusilliEnv
cd FusilliEnv
source bin/activate
```

Installing Pyaella

Install Pyaella directly from Github (the prefered method), or download and copy the Pyaella egg to your new FusilliEnv project directory. Make sure you are still using the python executables in your new virtualenv (pip, easy_install, etc..)

Github

```
git clone http://github.com/migacollabs/Pyaella@master
```

Github into a virtualenv

```
pip install git+https://github.com/migacollabs/Pyaella@master
```

From an .egg

```
easy_install Pyaella[most current version].egg
```

Please refer to Installing Pyaella for more details

Make a new application

Using the python of the virtualenv, create a new demo application called Fusilli

```
python -m pyaella.scaffold.build Fusilli
```

Note: The directory structure of an app with a virtualenv can seem overly nested. However, each level has a purpose, and more details on this are included in the rest of the docs. But briefly:

VirtualEnvRoot

AppDevRoot(Git)

AppRoot AppPackage (The App)

---

export the PYTHONPATH to the new Fusilli **AppRoot**

```
export PYTHONPATH=[/Users/..../FusilliEnv/Fusilli/fusilli
```

Warning: This PYTHONPATH export is very important. Nothing is expected to work if this is accidentally skipped

After running the scaffolding command, an application directory structure and initial files are created.

```
FusilliEnv/
  Fusilli/
    setup.py
    requirements.txt
    CHANGES.txt
    README.txt
    fusilli/
      development.ini
      production.ini
      fusilli/
        appcfg.yaml
        bgprocs.py
        composites.py
        contexts.py
        views.py
        assets/
        css/
        depot/
        player/
        sass/
        scripts/
        templates/
        uploads/
```

Included in this scaffold build are the standard javascript, coffeescript, sass, and Foundation libraries supported by Pyaella and available to Fusilli. The application is set up for immediate editing.

Create a new Domain file

Note: If you received an example domain file from Miga copy it into the AppPackage... or...

Create a new domain file, saving it in the AppPackage. Then compile the domain using *pyaellac*, and save a new models module in the AppPackage:

You can view the new models.py and see that it has generated some base classes. These can be edited, but usually during initial development the fast iterations of data or logical model changes are so rapid it is much easier to update the domain file and recompile the models.

Edit the application config

Now we can edit the appcfg.yaml file in the AppPackage, setting the correct Database values to point to your new localhost or remote test database.

```
Resources:
    Database:
        User: postgres
        Password:
        Host: localhost
        Port: 5432
        Schema: fusilli
        CreateTables: True
```

When starting up Pyaella or *dashi* will connect to this host and schema using the credentials you supply. If the machine is not accessible or you don't have the proper credentials it will fail.

Testing the models and connection to the database

Once the entire environment is set up and the application has been built, you can use *dashi* to interact with your models and the database. The following command will start up an interactive that lets you play with your new models. If *dashi* throws an exception you know there is problem with the connection, the generated code, the PYTHONPATH is incorrect, or there is a conflict in your environment.

```
python -i -m pyaella.orm.dashi -c fusilli/appcfg.yaml
```

class pyaella.**Configures**
marking type

class pyaella.**Configurable**
marking type

class pyaella.**Affector**
class that supports context manager, planner

class pyaella.**Affectable**
marking type

class pyaella.**Mix**
marker interface to Mix in Mixables

class pyaella.**Mixable**

class pyaella.**Container**(*klass*, ***kwds*)
an object that dynamically collects and sets attribute from a class definition and keywords on __init__.

future: supporting exclusion of attributes passed based on name

class pyaella.**SynchronisedContainer**

pyaella.**register_decorator**(*decorator*)

pyaella.**memoize**(*func*)

pyaella.**memoize_exp**(*expiration=None*)
Memoize with an expiration time (in seconds)

pyaella.**argument_bind_params**(*func*)
@argument_bind_params

class pyaella.**ExpFilter**(**args*, ***kwds*)

class Happy(ExpFilter): pass

class Sad(ExpFilter):

def __init__(self, who, what, where, when, why): ExpFilter.__init__(self, who, what, where, when, why)

class TestExpressible(object): @accepts_expfilter def __mod__(self, other, **kwds):

def happy_func(*args):

print 'Happy Func called', dir(args[0]), args[0]._kwds, args[0]._args

def sad_func(*args):

print 'Sad Func called', dir(args[0]), args[0]._kwds, args[0]._args

dispatch = { 'Happy':happy_func, 'Sad':sad_func

} try:

dispatch[kwds.keys()[0]](kwds[kwds.keys()[0]])

except KeyError: raise

TestExpressible() % Sad(1,2,3,4,5)

pyaella.**recordtype**(*typename*, *field_names*, *verbose=False*, ***default_kwds*)

Returns a new class with named fields.

@keyword field_defaults: A mapping from (a subset of) field names to defaultvalues.

@keyword default: If provided, the default value for all fields without an explicit default in *field_defaults*.

```
>>> Point = recordtype('Point', 'x y', default=0)
>>> Point.__doc__           # docstring for the new class
'Point(x, y)'
>>> Point()                 # instantiate with defaults
Point(x=0, y=0)
>>> p = Point(11, y=22)     # instantiate with positional args or keywords
>>> p[0] + p.y              # accessible by name and index
33
>>> p.x = 100; p[1] =200    # modifiable by name and index
>>> p
Point(x=100, y=200)
>>> x, y = p               # unpack
>>> x, y
(100, 200)
>>> d = p.todict()         # convert to a dictionary
>>> d['x']
100
>>> Point(**d) == p        # convert from a dictionary
True
```

1.1.3 Interactive Pyaella using Dashi

Using *dashi* you can start an interactive session to query, browse, experiment and run tests... as well as full Create, Read, Update, Delete depending on your role set in the database.

Dashi doesn't start an interactive shell in which to run by itself, so it must be started with the *-i* python argument.

To start a Dashi run, you can *cd* into a Pyaella project, making sure to have the correct *PYTHONPATH* exported, and run a command in a shell such as this:

python -i -m pyaella.orm.dashi -c mysite/app.yaml

The first argument *-i* puts us in interactive mode, so the Python interpreter will not exit at the end of its entry point. The *-m* argument sets the entry point of the interpreter, which in this case is Dashi. We provide Dashi a Pyaella application configuration file, which describes the application, where to find the Pyaella *domain* files for the ORM, etc..

Dashi will set up the environment, dynamically load the compiled Pyaella data models, inject the schemas, reflect anything specified, and set up the SQLAlchemySessionFactory.

Getting a Session

By calling *get_session()* you can get a new Session object, which is already attached to a connection in the connection pool, which is connected to your databased specified in the application configuration file supplied by the *-c* command line argument.

s = get_session()

This is a full SQAlchemy Session object and can perform all its normal and expected functions.

Working with logical models

Logical models and physical data models are different things. Logical models describe things in an abstract, decoupled way more suited to logical expression and execution. Physical ata models define how attributes and contraints are created and stored in persistence layers. (This is the Fisher Price explanation, of course.)

Here we define *logical* models as *logical data models* and *data models* as *physical data models*... an abbr per Data Models.

The logical models in Pyaella are dynamically created from metaclasses, using dependency injection from a *domain* or schema file. These models are accessible in Dashi by the dynamically created model namespace *models*. Any model described in the domain file is then accessible in this module.

asset = models.Asset

Pyaella models are logic objects, not pure data models, and so the underlying or canonical ORM state of a Pyaella model is interacted with using the tilde ~ operator. More of logic objects, state, and ORM capabilities of Pyaella object are described (here).

A simple query for an Asset

```
s = get_session()
q = s.query(~models.Asset).filter((~models.Asset).asset_id==6678036)
rp = q.all()
for asset in rp:
    print asset.asset_id
```

Here we get a session from the SQLAlchemySessionFactory using *get_session()*. Using this Session we create a query for Assets, filtering for Assets that have the *asset_id* 6678036.

When we run the query, asking for *all()* results, we receive a *ResultProxy*, which is a collection that emulates a python *list*, so we can iterate over the results.

Of course we could have run the query and asked for just the 'first()', and received a *ResultProxy* for just the one row, as the asset_id in this example is the primary key of this table.

A quick example of a single JOIN query coded in a typical Pyaella style, 'prototyping' the Pyaella model's ORM mapping with the ~ canonical access operator for readability.

```
# get a session
s = get_session()

A, AL = ~models.Asset, ~models.AssetLoction

# create query, joining Asset with AssetLoction
rp = (s.query(A, AL)
        .filter(A.asset_id==6678036)).all()

# for each now in the result
# unpack the tuple row
for row in rp:
    asset, asset_location = row
```

Pyaella model characteristics, introspection

A Pyaella model has methods to help describe it to the world, namely to help work with its associated Table in a database if it has one, its columns and types, constraints.

It does this by working through the underlying SQAlchemy objects, and exposes them for directly business logic manipulation and expressions.

Getting the FieldDef object from an Asset object and printing its string representation would display the Column definition by SQLAlchemy. However, the FieldDef object has many methods of its own to work or express the *fields* of the Pyaella entity, which may or may not be directly associated to a database, table or view.

```
>>>fld_df = asset.field_def('asset_id')
Column(BIGINT, nullable=False, unique=None, primary_key=True)
```

Example of a Dashi run

Here we get a session from the *SQLAlchemySessionFactory*, and after getting the mapped class for an AssetLocation query for a row in the *asset_locations* table.

The object returned is a normal SQLAlchemy object, attached to the session, which is attached to its underlying connection pool.

We can create a Pyaella entity by passing the ORM object into the *__init__*, which allows us to get properties and attributes specified by the application's *domain*.

```
>>> s = get_session()
>>> AL = ~models.AssetLocation
>>> rp = s.query(AL).filter(AL.asset_location_id==8517839).all()
>>> rp
[<pyaella.orm.AssetLocation object at 0x1057acc10>]
>>> al = models.AssetLocation(entity=rp[0])
>>> al
<saimin.models.AssetLocation object at 0x1057a9ed0>
>>> al.PrimaryKeyName
u'asset_location_id'

>>> for field in al.Fields:
...     field
...
u'asset_location_id'
u'path_id'
u'asset_id'
u'folder_from_path'
u'alt_filename'
u'initial_entry_by'
u'initial_entry_date'
u'last_upd'
u'last_uid'
u'last_operation'
u'qc_status_id'
u'archive_status'
```

Every Pyaella *entity* can be converted to a standard python dictionary using the *to_dict()* method.

```
>>> pprint.pprint(d)
{u'alt_filename': None,
 u'archive_status': u'ON',
 u'asset_id': 6677924L,
 u'asset_location_id': 8517839L,
 u'folder_from_path': u'_4/5000/4548/',
 u'initial_entry_by': u'mr_jobs_godzilla',
 u'initial_entry_date': datetime.datetime(2012, 9, 5, 13, 13, 1, 895192),
 u'last_operation': u'UPDATE',
```

```
 u'last_uid': u'mr_jobs_godzilla',
 u'last_upd': datetime.datetime(2012, 9, 22, 22, 31, 35, 720669),
 u'path_id': 3,
 u'qc_status_id': 4}
```

But that doesn't necessarily mean that the data is 'jsonable', Pyaella's default and preferred method of data representation. In this example the *json* modules throws an exception when attempting to serialize a python datetime object.

```
>>> json.dumps(d)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/opt/local/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/json/__init__.py",
    return _default_encoder.encode(obj)
  File "/opt/local/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/json/encoder.py", 1
    chunks = self.iterencode(o, _one_shot=True)
  File "/opt/local/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/json/encoder.py", 1
    return _iterencode(o, 0)
  File "/opt/local/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/json/encoder.py", 1
    raise TypeError(repr(o) + " is not JSON serializable")
TypeError: datetime.datetime(2012, 9, 22, 22, 31, 35, 720669) is not JSON serializable
```

However, this is not an issue for most common object types supported by Pyaella. Each model also has a *__json__* special method to serialize an *entity* appropriately. This method also takes a *webob request* object as an argument, but in this example we can pass a *None*.

```
>>> al.__json__
<bound method AssetLocation.__json__ of <saimin.models.AssetLocation object at 0x1057a9ed0>>
>>> al.__json__(None)
{u'asset_id': 6677924L, u'folder_from_path': u'_4/5000/4548/', u'last_operation': u'UPDATE', u'archiv
```

1.1.4 Domain files

Pyaella uses a *domain* file to describe an application or system's logical models. The domain file is compiled by *pyaella.metacode.pyaellac* into python modules that can be used in conjunction with the domain file and any of its dependency injection declarations, attributes, or imports.

Here we define a *User* logic model than will be used to dynamically create a business object and an underlying SQLAlchemy ORM object.

```
User:
    Fields:
        id: Column(BigInteger,
            Sequence('users_id_seq', start=1011011011),
            CheckConstraint('id<11111111111'),
            server_default=text("nextval('users_id_seq')"),
            primary_key=True)
        user_name: Column(String(50), nullable=False, unique=True)
        email_address: Column(String(255), nullable=False, unique=True)
        password: Column(String(128),
            CheckConstraint('"char_length"(password)>126'), nullable=False)
        is_active: Column(Boolean, default=True)
        first_name: Column(String(64), nullable=False)
        last_name: Column(String(64), nullable=False)
        country_code: Column(String(3), nullable=False)
        phone_number: Column(String(15))
        address1: Column(String(255))
        address2: Column(String(255))
```

```
        city: Column(String(255))
        region: Column(String(255))
        post_code: Column(String(16))
        open_id: Column(String(128))
        auth_code: Column(String(8))
        access_token: Column(String(128))
        device_tokens: Column(Text)
```

Each field in the *Fields* attribute can become a SQLAlchemy *Column*, mapped to a *users* table in the database. If a field is not a SQLAlchemy construct, its created as a Python *slot*.

We can add some object and table spanning Rules to this logic model:

```
.
.
.
        auth_code: Column(String(8))
        access_token: Column(String(128))
        device_tokens: Column(Text)
    Rules:
        Unique: [user_name, email_address]
```

We can also define some custom actions and options:

```
.
.
.
    Options:
        - add_tr_standard_mod.sql.mako
```

And if this logic model is destined to be persisted in a table in a database, we can run literal SQL statements directly after the table is created:

```
.
.
.
    SQL: |

        INSERT INTO users
            (   user_name, email_address, password,
                first_name, last_name, country_code,
                is_active,
                access_token
            )
        VALUES
            (   'xef', 'el.xef@miga.me', '12345678',
                'Xef', 'Fe', 'USA',
                False
            );
```

Once the Pyaella logical model is defined in the domain file and compiled by *pyaellac*, the resulting models module can be run directly by python, which will instantiate each model, and have SQLAlchemy emit the SQL statements to stdout:

```
CREATE TABLE users (
    key VARCHAR,
    initial_entry_by VARCHAR,
    initial_entry_date TIMESTAMP WITHOUT TIME ZONE,
    last_uid VARCHAR,
    last_opr VARCHAR,
    last_upd TIMESTAMP WITHOUT TIME ZONE,
```

```
    phone_number VARCHAR(15),
    city VARCHAR(255),
    first_name VARCHAR(64) NOT NULL,
    last_name VARCHAR(64) NOT NULL,
    open_id VARCHAR(128),
    access_token VARCHAR(128),
    address1 VARCHAR(255),
    address2 VARCHAR(255),
    is_active BOOLEAN,
    post_code VARCHAR(16),
    country_code VARCHAR(3) NOT NULL,
    auth_code VARCHAR(8),
    region VARCHAR(255),
    device_tokens TEXT,
    password VARCHAR(128) NOT NULL CHECK ("char_length"(password)>126),
    email_address VARCHAR(255) NOT NULL,
    id BIGINT DEFAULT nextval('users_id_seq') NOT NULL CHECK (id<11111111111),
    user_name VARCHAR(50) NOT NULL,
    PRIMARY KEY (id),
    UNIQUE (user_name, email_address),
    UNIQUE (email_address),
    UNIQUE (user_name)
)
```

However, this SQL statement emmision isn't needed or required, but gives the developer options to either create databases and tables 'on-the-fly' with Pyaella, or to have Pyaella and SQLAlchemy reflect the database upon each start.

More of the domain-specific syntax of Pyaella and its domain file will be explained else where in this documentation.

More complete example

The following is a fairly complete example of a domain file that describes *Users* in a fairly typical web application:

```
PyaellaConfig:
    DeclBase: PyaellaSQLAlchemyBase
    MetaCls: PyaellaDataModelMetaclass
    ReflCls: PyaellaReflectiveModelMetaclass
    SuperCls: PyaellaDataModel

ApplicationDomain:
    Fields:
        id: Column(BigInteger,
            Sequence('application_domains_id_seq', start=1357913579135791),
            CheckConstraint('id<2468024680246802'),
            server_default=text("nextval('application_domains_id_seq')"),
            primary_key=True)
        name: Column(String(64), unique=True, nullable=True)
    Options:
        - add_tr_standard_mod.sql.mako

Application:
    Fields:
        id: Column(Integer,
            Sequence('applications_id_seq', start=1600),
            CheckConstraint('id<2600'),
            server_default=text("nextval('applications_id_seq')"),
            primary_key=True)
```

```
        name: Column(String(64), unique=True, nullable=True)
        key_sequence: Column(String(36))
    Relations:
        application_domain_id: Column(BigInteger, ForeignKey('application_domains.id'))
    Rules:
        Unique: [id, application_domain_id]
    Options:
        - add_tr_standard_mod.sql.mako

User:
    Fields:
        id: Column(BigInteger,
            Sequence('users_id_seq', start=1011011011),
            CheckConstraint('id<11111111111'),
            server_default=text("nextval('users_id_seq')"),
            primary_key=True)
        user_name: Column(String(50), nullable=False, unique=True)
        email_address: Column(String(255), nullable=False, unique=True)
        password: Column(String(128),
            CheckConstraint('"char_length"(password)>126'), nullable=False)
        is_active: Column(Boolean, default=True)
        first_name: Column(String(64), nullable=False)
        last_name: Column(String(64), nullable=False)
        country_code: Column(String(3), nullable=False)
        phone_number: Column(String(15))
        address1: Column(String(255))
        address2: Column(String(255))
        city: Column(String(255))
        region: Column(String(255))
        post_code: Column(String(16))
        open_id: Column(String(128))
        auth_code: Column(String(8))
        access_token: Column(String(128))
        device_tokens: Column(Text)
    Rules:
        Unique: [user_name, email_address]
    Options:
        - add_tr_standard_mod.sql.mako
    SQL: |

        INSERT INTO users
            (   user_name, email_address, password,
                first_name, last_name, country_code,
                is_active
            )
        VALUES
            (   'xef', 'el.xef@miga.me', '12345678',
                'Xef', 'Fe', 'USA',
                False
            );

UserTypeLookup:
    Fields:
        name: Column(String(24), unique=True, nullable=False)
        description: Column(String(64), nullable=False)
    Values:
        name: [
            Sys,
```

```
            Dev,
            Admin,
            General,
            Bot
        ]
        description: [
            System Administrator,
            Developer,
            Administrator,
            General User,
            Robot
        ]
    Options:
        - add_tr_standard_lu_mod.sql.mako

UserXUserTypeLookup:
    Relations:
        user_id: Column(BigInteger, ForeignKey('users.id'))
        user_type_id: Column(Integer, ForeignKey('user_type_lu.id'))
    Rules:
        Unique: [user_id, user_type_id]
    Options:
        - add_tr_standard_lu_mod.sql.mako

UserAppLicense:
    Fields:
        app_id: Column(Integer, nullable=False)
        domain_id: Column(BigInteger, nullable=False)
        start: Column(DateTime)
        expiry: Column(DateTime)
        license_hash: Column(String(256))
    Relations:
        user_id: Column(BigInteger, ForeignKey('users.id'), nullable=False)
    Rules:
        Unique: [app_id, domain_id, user_id]
    Options:
        - add_tr_standard_mod.sql.mako

Group:
    Fields:
        name: Column(String(24), nullable=False)
        description: Column(String(128), nullable=False)
        display_name: Column(String(64), nullable=False)
    Rules:
        Unique: [name, application_group_type_id]
    Relations:
        application_group_type_id: Column(Integer, ForeignKey('application_group_type_lu.id'))
    Options:
        - add_tr_standard_mod.sql.mako

ApplicationGroupTypeLookup:
    Fields:
        name: Column(String(24), unique=True, nullable=False)
        description: Column(String(64), nullable=False)
    Values:
        name: [
            Mividio RS,
            Mividio Personal
```

```
        ]
        description: [
            Real Estate Mividio Application,
            Personal Mividio Application
        ]
    Options:
        - add_tr_standard_lu_mod.sql.mako

UserXGroup:
    Relations:
        user_id: Column(BigInteger, ForeignKey('users.id'))
        group_id: Column(Integer, ForeignKey('groups.id'))
    Rules:
        Unique: [user_id, group_id]
    Options:
        - add_tr_standard_mod.sql.mako

UserTeam:
    Fields:
        name: Column(String(24), nullable=False)
    Relations:
        team_manager_id: Column(BigInteger, ForeignKey('users.id'), nullable=False)
    Rules:
        Unique: [name, team_manager_id]
    Options:
        - add_tr_standard_mod.sql.mako

UserXUserTeam:
    Relations:
        user_id: Column(BigInteger, ForeignKey('users.id'))
        user_team_id: Column(BigInteger, ForeignKey('user_teams.id'))
    Rules:
        Unique: [user_id, user_team_id]
    Options:
        - add_tr_standard_mod.sql.mako

Country:
    Fields:
        order_id: Column(Integer)
        common_name: Column(String)
        formal_name: Column(String)
        type: Column(String)
        sub_type: Column(String)
        sovereignty: Column(String)
        capital: Column(String)
        ISO_4217_currency_code: Column(String(3))
        ISO_4217_currency_name: Column(String)
        ITU_T_telephone_code: Column(String(3))
        ISO_3166_1_2_letter_code: Column(String(3))
        code: Column(String(3))
        ISO_3166_1_number: Column(String(3))
        IANA_country_Code_TLD: Column(String(3))

Asset:
    Fields:
        id: Column(Integer,
            Sequence('assets_id_seq', start=10000),
            server_default=text("nextval('assets_id_seq')"),
```

```
            primary_key=True)
        name: Column(String(64))
        description: Column(Text)
        asset_name: Column(String(256), nullable=False)
        height: Column(Integer, nullable=False)
        width: Column(Integer, nullable=False)
        filesize: Column(BigInteger, nullable=False)
        extention: Column(String(3), nullable=False)
        date_uploaded: Column(DateTime, default=datetime.datetime.now)
    Relations:
        owner_id: Column(BigInteger, ForeignKey('users.id'), nullable=False)
    Options:
        - add_tr_standard_mod.sql.mako

REFLECTIVE:
    CountryLookup:

AFTER_CREATE_SQL:
    Literal: |
```

1.1.5 Entity Request API

Indices and tables

- genindex
- modindex
- search

p

A

C

E

M

P

R

S